Natnael Kebede

TensorFlow is a very strong framework for create neural network

Question 1:

In the report, write comments for each line of code given above and explain what this framework is doing.

```
# import tensorflow library
import tensorflow as tf

# tensorflow library has many datasets, we will work with mnist
# MNIST handwritten digits dataset
mnist = tf.keras.datasets.mnist

# using load_data() function, we can load data for further work
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# All images have 256 colors (from 0 to 255, where 0 - black, 255 - white)
# But for any model, working with large numbers is more difficult
# because it's hard to find the necessary regularizers
# as a result, we normalize data
# Now, all numbers are from 0 to 1
x_train, x_test = x_train/255.0, x_test/255.0

# Creating neural network
# We create our own NN, for this, first we must create object which can keep all layers (Sequential).
# Then add Flatten() layers - flatten our feature map into a column. The flattening step is that we have long vector of input data that we processed further.
# The next is Dense layers. Dense implements the operation: output = activation(dot(input, kernel) + bias) where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer
# Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting.
# And finale Dense with 10 result

model = tf.keras.models.Sequential([
   tf.keras.layers.Flatten(),
   tf.keras.layers.Dense(512, activation = tf.nn.relu),
   tf.keras.layers.Dropout(0.2),
   tf.keras.layers.Dense(10, activation = tf.nn.softmax)
])

# Compile our model with Adam optimizer and sparse_categorical_crossentropy losses
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# training model
model.fit(x_train, y_train, epochs=5)

# Returns the loss value and metrics values for the model in test mode
model.evaluate(x_test, y_test)
```

Running the above frame work and plotting x_train[0] will give us

```
# import tensorflow library
import tensorflow as tf
```

In [2]:

```
# tensorflow library has many datasets, we will work with mnist
# MNIST handwritten digits dataset
mnist = tf.keras.datasets.mnist
```

In [3]:

```
# using load_data() function, we can load data for further work
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

In [4]:

```
# we have 60000 train images with size 28x28, and 10000 test images
x_train.shape, x_test.shape
```

Out[4]:

```
((60000, 28, 28), (10000, 28, 28))
```

In [5]:

```
# example of data
x_train[0][:10]
```

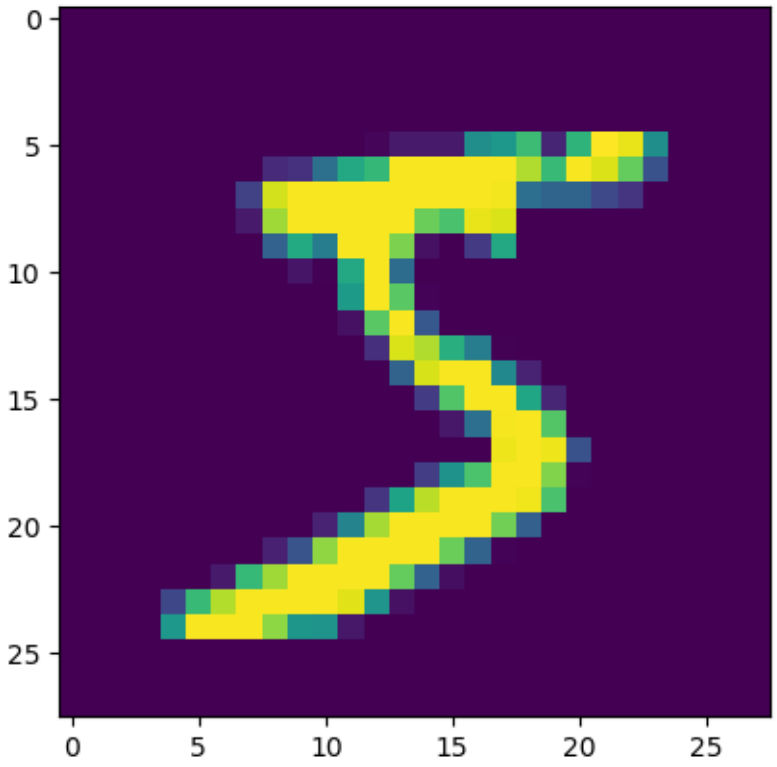
Out[5]:

```
array([[  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   0,   3,
         18,  18,  18, 126, 136, 175,  26, 166, 255, 247, 127,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  30,  36,  94, 154, 170,
        253, 253, 253, 253, 253, 225, 172, 253, 242, 195,  64,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  49, 238, 253, 253, 253, 253,
        253, 253, 253, 253, 251,  93,  82,  82,  56,  39,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,  18, 219, 253, 253, 253, 253,
        253, 198, 182, 247, 241,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0],
       [  0,   0,   0,   0,   0,   0,   0,   0,  80, 156, 107, 253, 253,
        205,  11,   0,  43, 154,   0,   0,   0,   0,   0,   0,   0,   0,
          0,   0]], dtype=uint8)
```

In [13]:

```
# display image
```

```
from matplotlib import pyplot as plt
plt.imshow(x_train[0])
plt.show()
```



In [7]:

```
#how we can see, this digit is 5. And y_train[0] must be 5
y_train[0]
```

Out[7]:

5

In [8]:

```
# All images has 256 colors (from 0 to 255, where 0 - black, 255 - white)
# But for any model, working with large numbers is more difficult
# because it's hard to find the necessary regularizers
# So we normalize data
# Now, all numbers are from 0 to 1
x_train, x_test = x_train/255.0, x_test/255.0
```

In [9]:

```
# Creating neural network
# We create our own NN, for this, firstly we must create object which can keep all layers
(Sequential).
# Then add Flatten() layers - flatten our feature map into a column. The flattening step
is that we have long vector of input data that we processed further.
# The next is Dense layers. Dense implements the operation: output = activation(dot(input
, kernel) + bias) where activation is the element-wise activation function passed as the
```

```
activation argument, kernel is a weights matrix created by the layer, and bias is a bias
vector created by the layer

# Dropout consists in randomly setting a fraction rate of input units to 0 at each update
during training time, which helps prevent overfitting.

# And finale Dense with 10 result


model = tf.keras.models.Sequential([

    tf.keras.layers.Flatten(),

    tf.keras.layers.Dense(512, activation = tf.nn.relu),

    tf.keras.layers.Dropout(0.2),

    tf.keras.layers.Dense(10, activation = tf.nn.softmax)

])
```

In [10]:

```
# Compile our model with Adam optimizer and sparse_categorical_crossentropy losses

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accurac
y'])
```

In [11]:

```
# training model

model.fit(x_train, y_train, epochs=5)
```

```
Epoch 1/5
60000/60000 [==============================] - 37s 617us/step - loss: 0.1989 - acc: 0.941
4
Epoch 2/5
60000/60000 [==============================] - 37s 619us/step - loss: 0.0793 - acc: 0.975
5
Epoch 3/5
60000/60000 [==============================] - 32s 531us/step - loss: 0.0508 - acc: 0.984
6
Epoch 4/5
60000/60000 [==============================] - 35s 576us/step - loss: 0.0377 - acc: 0.988
1
Epoch 5/5
60000/60000 [==============================] - 34s 569us/step - loss: 0.0268 - acc: 0.991
7
```

Out[11]:

```
<tensorflow.python.keras.callbacks.History at 0x1bc65c8beb8>
```

In [12]:

```
# Returns the loss value & metrics values for the model in test mode

model.evaluate(x_test, y_test)
```

```
10000/10000 [==============================] - 1s 147us/step
```

Out[12]:

```
[0.06532145720738045, 0.9804]
```

So, we create NN and obtain 0.9804 accuracy for test data¶

Once we know this, the features it has learned from training can be plotted below. This was done by reshaping the learned weights or vectors to the image dimension (in 2D) which is shown below. The number of hidden nodes in this case is 512.

```
# Using get_weights() and reshape() we can get and reshape the learned weigh
ts
# We have 512 2D vectors with size like images
model.get_weights()[0].reshape(512,28,28)
```

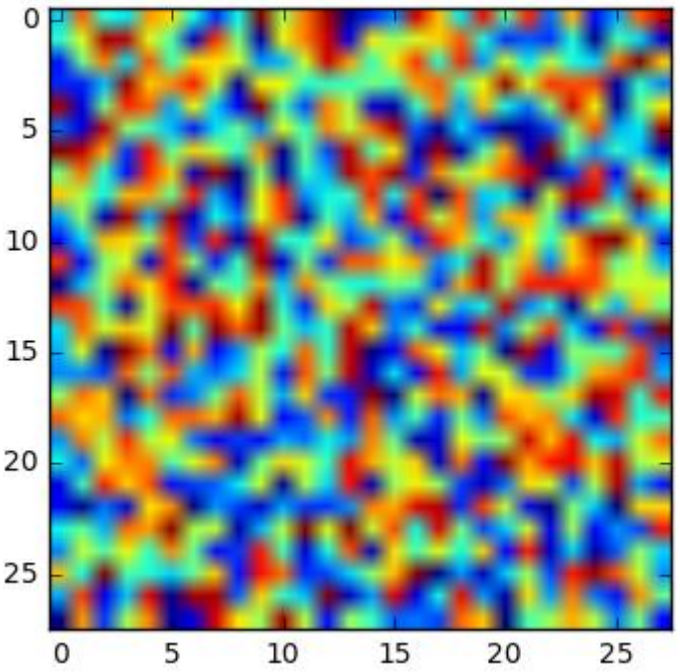
Out[42]:

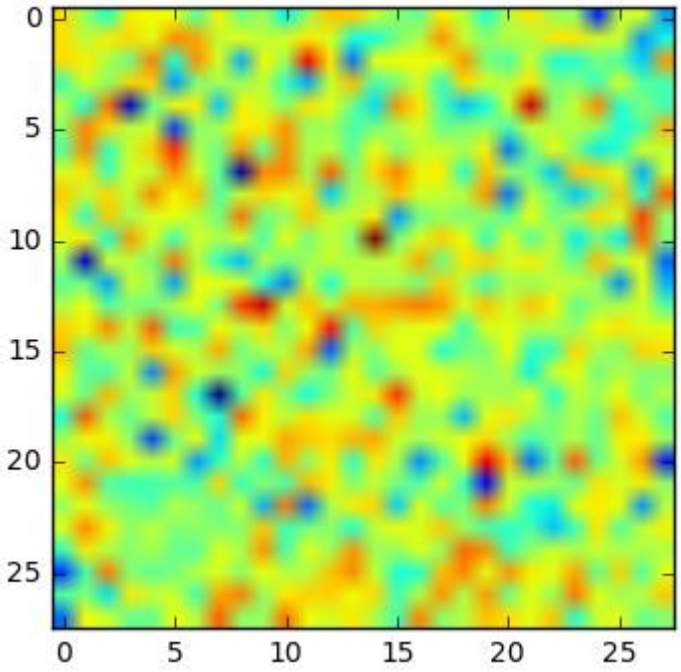
```
array([[[-0.02229665,  0.04139172, -0.01582594, ..., -0.02802414,
          0.04168969,  0.05916187],
        [-0.01287649,  0.01572127,  0.06306827, ..., -0.0087525 ,
         -0.02196766, -0.0582088 ],
        [-0.04825814, -0.00181364,  0.03818365, ...,  0.03946926,
          0.06792642,  0.02690922],
        ...,
        [ 0.02780189, -0.01659955,  0.06604363, ...,  0.0427405 ,
          0.01180129, -0.03139378],
        [-0.02779291,  0.04701054, -0.05465636, ..., -0.04833211,
          0.03449263, -0.003576  ],
        [-0.06550936,  0.03293993, -0.04334152, ..., -0.03403458,
         -0.0009474 , -0.0510423 ]],

       [[ 0.04987319, -0.0032774 ,  0.00819252, ...,  0.03825295,
         -0.03834459, -0.01040468],
        [-0.02446838,  0.01665464, -0.04315042, ...,  0.06253843,
         -0.01543855, -0.06192936],
        [ 0.03095727,  0.03620575,  0.01622684, ...,  0.04326359,
         -0.01284431,  0.03510014],
        ...,
        [-0.03480663, -0.00626779, -0.00619187, ..., -0.04476974,
          0.00619422, -0.02920216],
        [ 0.05126655, -0.00710559,  0.06079198, ..., -0.00122587,
         -0.04489604, -0.04889568],
        [-0.05895912, -0.01605516, -0.05487505, ..., -0.01115832,
          0.01307904, -0.01270497]],

       [[-0.06391908, -0.05837409, -0.06529132, ..., -0.00898878,
          0.03664443, -0.00978151],
        [-0.02975005,  0.06067292, -0.03737972, ...,  0.06377114,
          0.0612458 ,  0.05575173],
        [ 0.03498362,  0.05234125,  0.0519313 , ..., -0.01150131,
          0.05428535, -0.01960634],
        ...,
        [-0.06398433,  0.03044017, -0.01906756, ..., -0.04331888,
         -0.04788227, -0.01285047],
        [-0.04457147,  0.06151864,  0.00728152, ..., -0.05035496,
         -0.06721275, -0.04392593],
        [ 0.00028819,  0.05450474, -0.05663214, ...,  0.03418684,
          0.00919996,  0.01581306]],

       ...,

       [[-0.0528694 , -0.00700456,  0.05375338, ..., -0.02986717,
          0.03555479,  0.02847249],
        [-0.00839807, -0.02383403,  0.09543508, ..., -0.0020743 ,
          0.06411247,  0.00508043],
        [ 0.03235099, -0.08094167,  0.0081707 , ...,  0.02257293,
         -0.05831848, -0.0046307 ],
        ...,
        [-0.02553285, -0.00798951, -0.0350895 , ..., -0.02196366,
         -0.06508023,  0.00011905],
        [-0.0466458 , -0.06490665, -0.00217569, ..., -0.03677712,
         -0.02886347,  0.05450716],
        [ 0.03572504, -0.01342753,  0.05686118, ..., -0.01550414,
         -0.00243859,  0.06266695]],
```

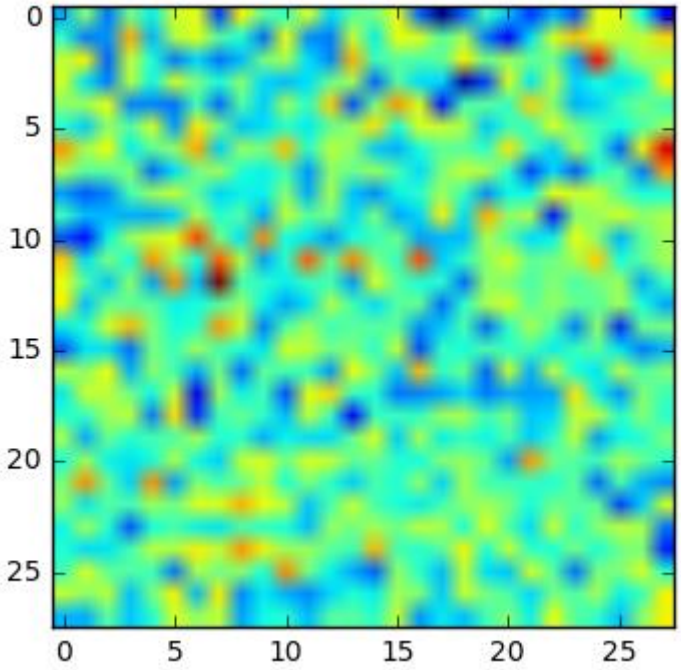
```
       [[-0.00169596,  0.02007296, -0.05462404, ...,  0.05288252,
         -0.04926099,  0.05109216],
        [ 0.04945004,  0.04215363,  0.00909673, ...,  0.023936  ,
          0.02381889, -0.03526631],
        [ 0.05909191,  0.02730811,  0.01774653, ...,  0.01028131,
          0.06040478, -0.06494322],
        ...,
        [-0.00492111,  0.05550149,  0.0296009 , ...,  0.05808868,
          0.02748642,  0.01567345],
        [-0.01806455, -0.00802287, -0.01305632, ...,  0.05868632,
         -0.01960167, -0.0091122 ],
        [-0.00668485,  0.04777103,  0.00252515, ...,  0.01024941,
         -0.01719204, -0.02978803]],

       [[-0.02365503, -0.01115865, -0.00151245, ..., -0.00315868,
          0.0424263 , -0.02329583],
        [ 0.04408326,  0.06035523,  0.05414539, ...,  0.02397346,
         -0.04115117,  0.03396166],
        [ 0.05656593, -0.06118607,  0.04426006, ...,  0.02952486,
          0.0491624 ,  0.01735847],
        ...,
        [-0.01246573, -0.02323239, -0.04509301, ...,  0.04170247,
          0.05555505,  0.06431068],
        [ 0.04082044, -0.00673017, -0.01398663, ...,  0.02759524,
          0.01944986,  0.04224379],
        [ 0.00680999,  0.02229011, -0.05478456, ..., -0.01883555,
         -0.05436125, -0.0216451 ]]], dtype=float32)
```

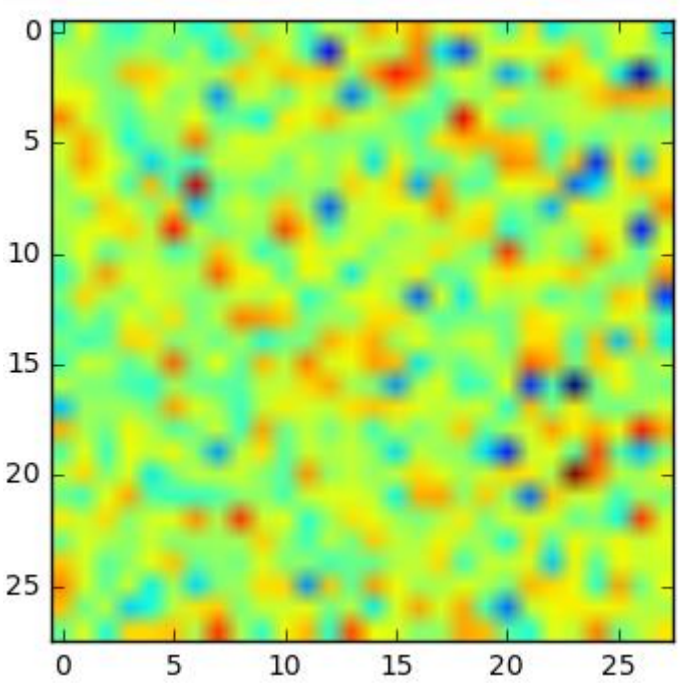
In [43]:

```
for i in range(0, 512, 100):
    plt.imshow(model.get_weights()[0].reshape(512,28,28)[i])
    plt.show()
```




0
5
10
15
20
25
0
5
10
15
20
25

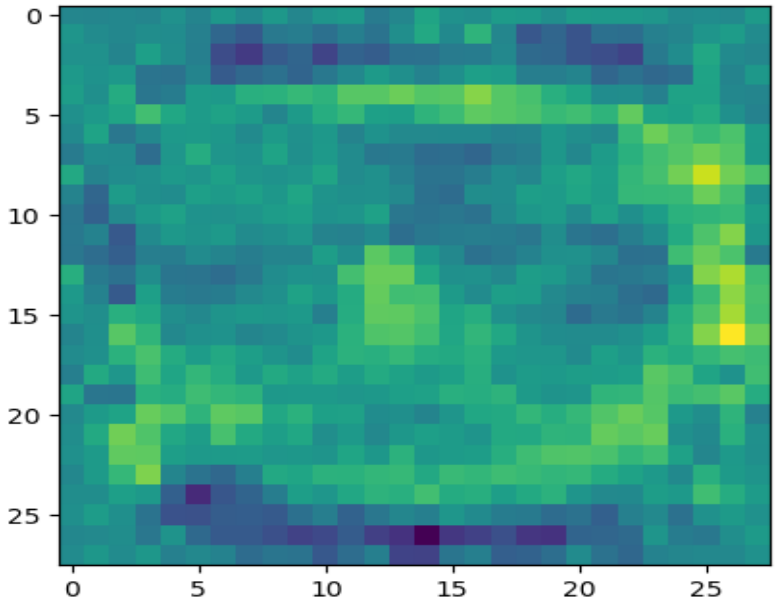

0
5
10
15
20
25
0
5
10
15
20
25

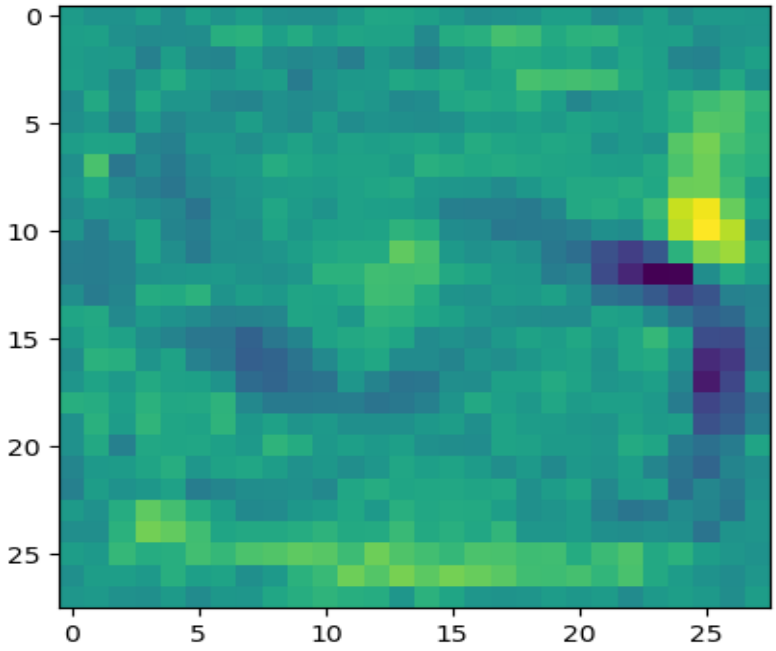


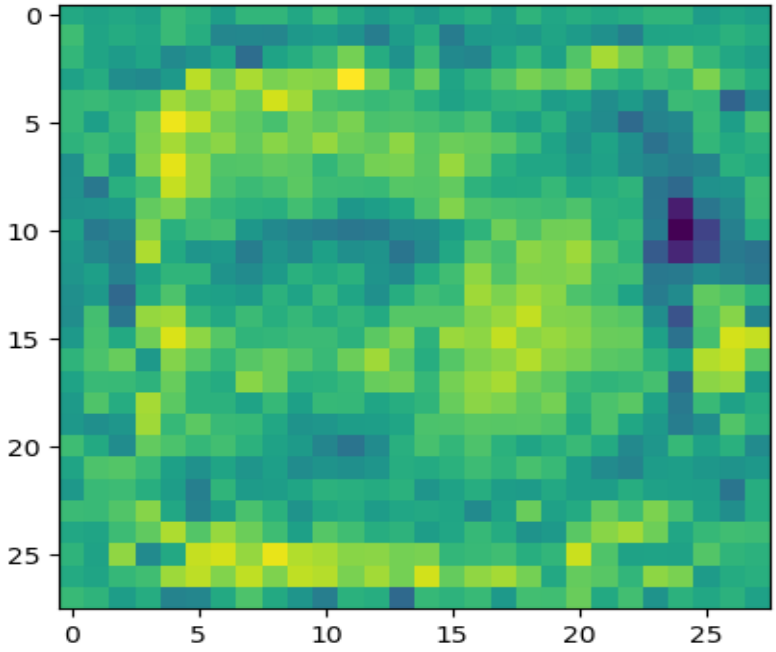
Question 2:

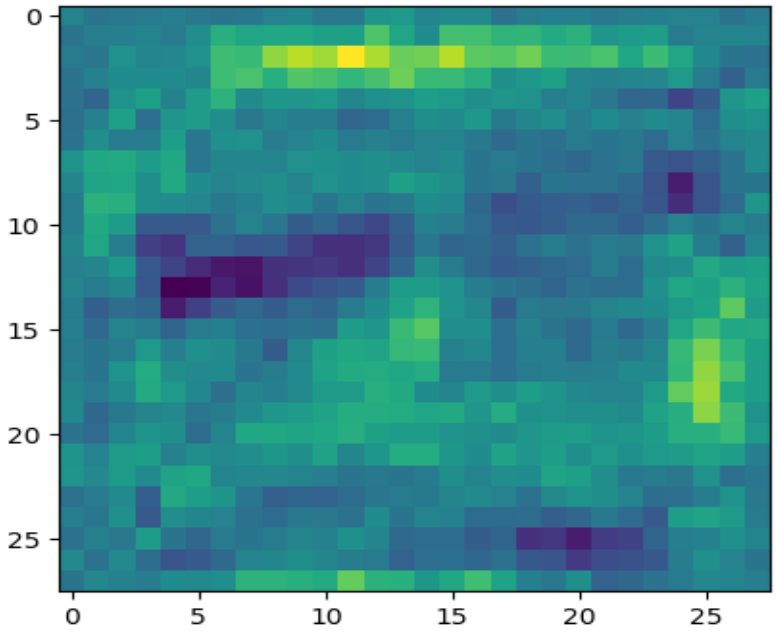
Change the number of hidden nodes to 10 and train this neural network. The trained model contains the weights that it has learned from training. Plot the features in the hidden layer that it has learned from training and include them in the report. That is, reshape the learned weights (vectors) in the first layer (between the input and the 1st hidden layer) to the image dimension (in 2D) and show them. (You will get 0 marks for this if the result is not included in the report.)

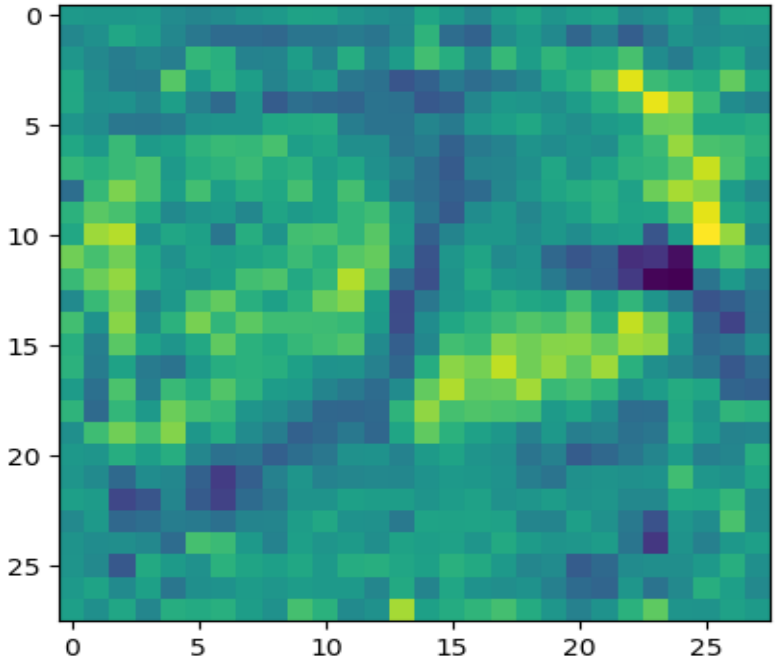
Doing the above procedure (now with 10 hidden nodes) and displaying the result of reshaping the learned weights(vectors) in the first layer (between the input and the 1st hidden layer) to the image dimension (in 2D) we have the following:

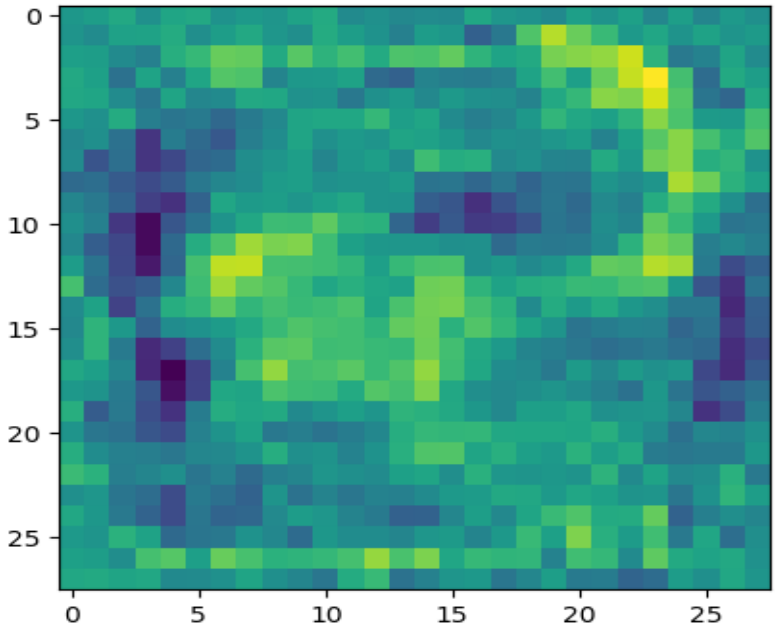


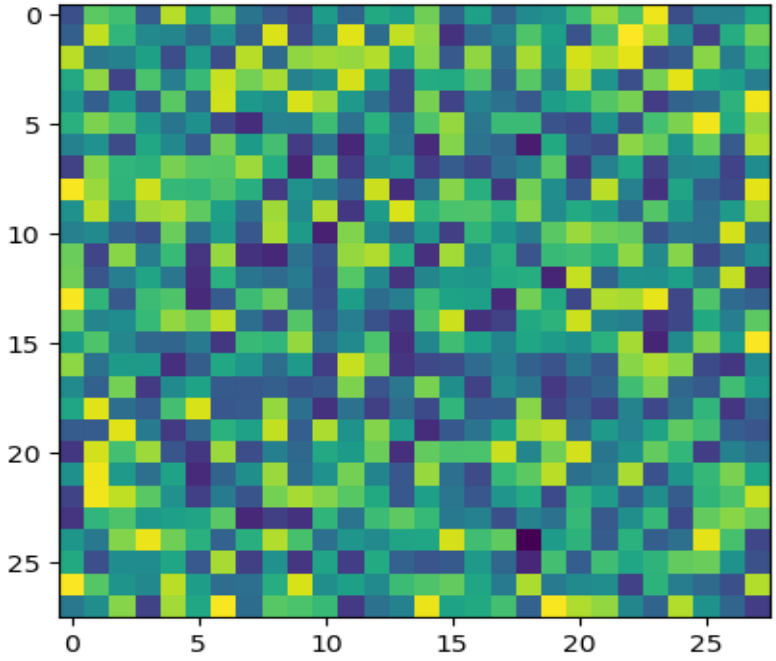

0
5
10
15
20
25
0
5
10
15
20
25

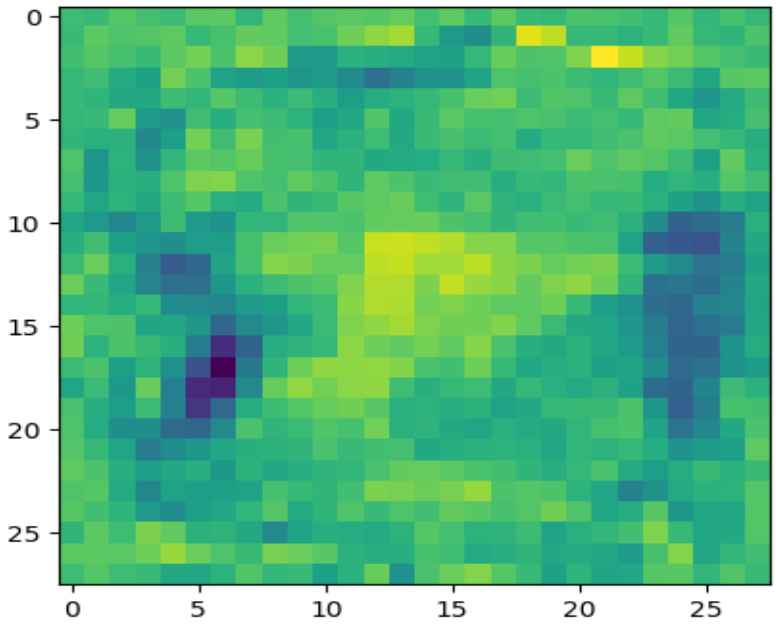

0
5
10
15
20
25
0
5
10
15
20
25

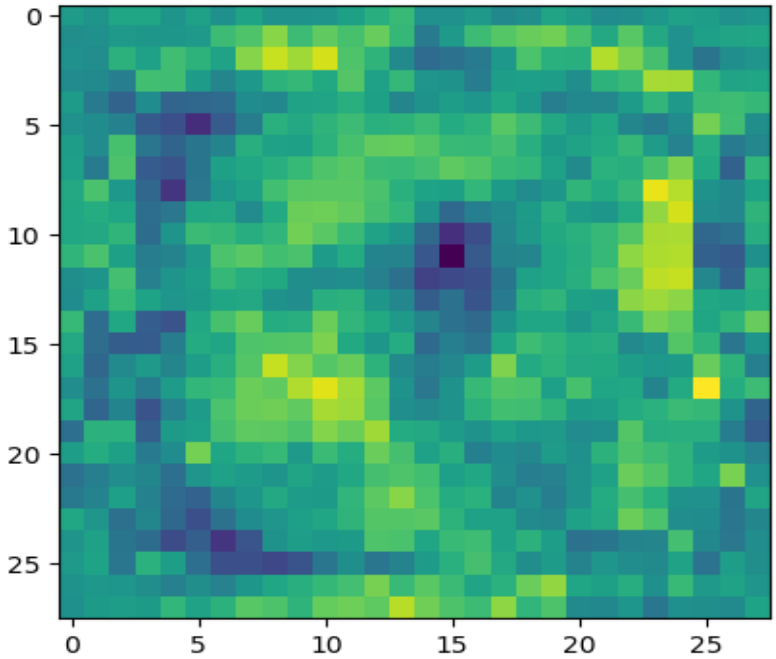

0
5
10
15
20
25
0
5
10
15
20
25

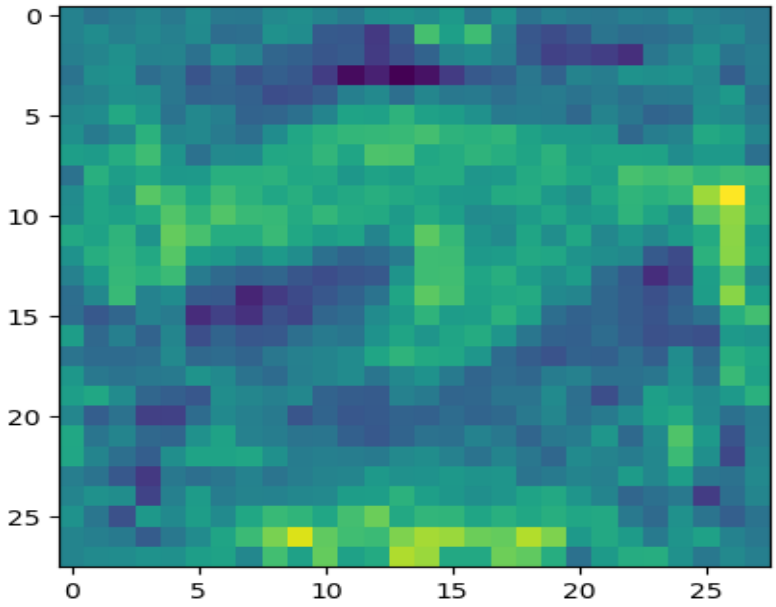

0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25



As expected, since there are 10 hidden nodes, we will get ten 2D figures. That can be explained by noticing that 28 x 28 = 784.

Question 3:

Change the number of hidden nodes to 1, 10, 50 and 100 and report how the testing accuracy changes for the testing dataset. Report the result and your observation in the report.

When number of hidden nodes = 1, we get the following dataset output of created NN and obtain 0.3299 (32.99%) accuracy for test data

```
Epoch 1/5
60000/60000 [==============================] - 3s 53us/step - loss: 1.9715 -
acc: 0.2263
Epoch 2/5
60000/60000 [==============================] - 2s 40us/step - loss: 1.7857 -
acc: 0.2821
Epoch 3/5
60000/60000 [==============================] - 2s 40us/step - loss: 1.7075 -
acc: 0.3009
Epoch 4/5
60000/60000 [==============================] - 2s 40us/step - loss: 1.6611 -
acc: 0.3173
Epoch 5/5
60000/60000 [==============================] - 2s 40us/step - loss: 1.6294 -
acc: 0.3270
10000/10000 [==============================] - 0s 36us/step
[1.6220794076919556, 0.3299]
```

When number of hidden nodes = 10, we get the following dataset output of created NN and obtain 0.9293 (92.93 %) accuracy for test data

```
Epoch 1/5
60000/60000 [==============================] - 3s 57us/step - loss: 0.5324 -
acc: 0.8486
Epoch 2/5
60000/60000 [==============================] - 3s 45us/step - loss: 0.2966 -
acc: 0.9155
```

```
Epoch 3/5
60000/60000 [==============================] - 3s 45us/step - loss: 0.2662 -
acc: 0.9238
Epoch 4/5
60000/60000 [==============================] - 3s 45us/step - loss: 0.2494 -
acc: 0.9284
Epoch 5/5
60000/60000 [==============================] - 3s 46us/step - loss: 0.2382 -
acc: 0.9327
10000/10000 [==============================] - 0s 40us/step
[0.2436472684428096, 0.9293]
```

When number of hidden nodes = 50, we get the following dataset output of created NN and obtain 0.9692 (96.92%) accuracy for test data

```
Epoch 1/5
60000/60000 [==============================] - 5s 78us/step - loss: 0.3257 -
acc: 0.9067
Epoch 2/5
60000/60000 [==============================] - 4s 60us/step - loss: 0.1584 -
acc: 0.9547
Epoch 3/5
60000/60000 [==============================] - 4s 60us/step - loss: 0.1178 -
acc: 0.9653
Epoch 4/5
60000/60000 [==============================] - 4s 59us/step - loss: 0.0954 -
acc: 0.9720
Epoch 5/5
60000/60000 [==============================] - 4s 60us/step - loss: 0.0814 -
acc: 0.9752
10000/10000 [==============================] - 0s 44us/step
[0.10414193934705109, 0.9692]
```

When number of hidden nodes = 100, we get the following dataset output of created NN and obtain 0.9752 (97.52%) accuracy for test data

```
Epoch 1/5
60000/60000 [==============================] - 7s 123us/step - loss: 0.2684 -
acc: 0.9244
Epoch 2/5
60000/60000 [==============================] - 7s 124us/step - loss: 0.1205 -
acc: 0.9649
Epoch 3/5
60000/60000 [==============================] - 7s 109us/step - loss: 0.0841 -
acc: 0.9751
Epoch 4/5
60000/60000 [==============================] - 7s 110us/step - loss: 0.0642 -
acc: 0.9803
Epoch 5/5
60000/60000 [==============================] - 7s 109us/step - loss: 0.0501 -
acc: 0.9844
10000/10000 [==============================] - 1s 54us/step
[0.08228329120108392, 0.9752]
```

We can observe two things here. First, the testing accuracy changes for each testing dataset. Additionally, the accuracy also changes every time we ran the code with the same hidden number of nodes since the data is randomly generated. Furthermore, with an increasing number of nodes (from 1 to 10 to 50 to 100), the observation shows us that the accuracy also increases (0.3299 or 32.99% to 0.9293 or 92.93 % to 0.9692 or 96.92 % to 0.9752 or 97.52%) respectively.